OPENSILICON

The Modern VLSI Layout IDE

"VSCode for Chip Design"

Development Plan & Technical Architecture

Version 1.0 — February 2026

CONFIDENTIAL

1. Executive Summary

OpenSilicon is a next-generation, open-source VLSI layout IDE designed to replace Microwind and compete with commercial EDA tools by bringing the extensibility, speed, and developer experience of modern code editors to the world of chip design. The core thesis is simple: IC layout design is fundamentally an editing and visualization problem, and EDA tools have ignored 20 years of advances in editor architecture, GPU rendering, and plugin ecosystems.

Microwind, while widely used in academia, suffers from a legacy single-threaded architecture, a rigid non-extensible UI, minimal keyboard-driven workflows, no version control integration, and outdated rendering. Engineers spend significant time fighting the tool rather than designing silicon.

OpenSilicon solves this by building on Electron/Tauri with a GPU-accelerated canvas, a command palette and keybinding system inspired by VSCode, a plugin architecture for custom DRC rules, PDKs, and simulation backends, and first-class support for modern workflows including Git integration, collaborative editing, and CI/CD for tapeout.

2. Problem Analysis: Why Microwind Falls Short

2.1 Pain Points for Users

- No extensibility: Users cannot add custom technology files, DRC decks, or simulation models without hacking the binary.
- Single-document interface: No tabs, no split views, no ability to cross-reference schematics and layout side by side.
- Primitive rendering: CPU-only rendering that stutters on designs beyond a few thousand polygons. No smooth zoom, no level-of-detail.
- No scripting or automation: Every action must be performed manually. No macro system, no batch processing, no headless mode.
- No version control: Binary .MSK file format with no diff/merge support. Collaboration requires emailing files.
- Rigid DRC: Hard-coded design rules with no user-configurable rulesets or multi-technology support.
- No dark mode or UI customization: Fixed UI with no theming, no configurable panels, no keyboard shortcuts.
- Limited simulation: Built-in SPICE is minimal. No integration with industry-standard simulators (ngspice, Xyce, Spectre).

2.2 What the Market Needs

The target users are university students, researchers, small-team ASIC startups, and engineers doing quick physical design exploration. They need a tool that is free or affordable, runs on all platforms, feels modern, supports real PDKs (like SkyWater SKY130, GF180MCU, IHP SG13G2), and integrates with the open-source EDA ecosystem (OpenROAD, Magic, KLayout, Yosys, netgen).

3. Product Vision & Core Principles

Principle	Description
Editor-First	The layout canvas is a code editor for geometry. Every action has a command, every command has a keybinding, every keybinding is customizable.
Extensible	PDKs, DRC rules, extraction scripts, simulation backends, and UI panels are all plugins. The core ships lean; the ecosystem provides power.
GPU-Native	All rendering via WebGPU/wgpu. Millions of polygons at 60fps. Semantic zoom with level-of-detail. Real-time DRC overlay.
Open Format	Native file format is human-readable (GDS-II + JSON metadata, or OASIS). Full Git diff/merge support. Import/export LEF/DEF, GDS-II, OASIS, CIF.
Integrated	Built-in terminal, Git panel, simulation waveform viewer, netlist editor, LVS comparison view. One tool, not ten.
Cross-Platform	Runs natively on Windows, macOS, and Linux with identical feature parity. Tauri-based for small binary size.

4. Technical Architecture

4.1 Application Shell: Tauri + React

The application shell uses Tauri (Rust backend) rather than Electron to achieve a significantly smaller binary (under 15MB vs 200MB+), lower memory usage, and native OS integration. The frontend is React with TypeScript, providing a component-based UI that mirrors VSCode's panel-based layout. The Rust backend handles all performance-critical operations: file I/O, geometry computation, DRC engine, and SPICE netlist parsing.

4.2 Rendering Engine: WebGPU Canvas

The layout canvas is powered by WebGPU (via wgpu-rs exposed to the frontend). This enables hardware-accelerated rendering of millions of polygons with per-layer coloring, transparency, hatching patterns, and real-time selection highlighting. The renderer implements semantic zoom: at high zoom levels, individual transistors show labels and pin names; at medium zoom, only metal routing and cell boundaries appear; at low zoom, a heatmap density view replaces individual shapes. A spatial index (R-tree) backed by the Rust layer provides O(log n) click-to-select and viewport culling.

4.3 Plugin System: WASM + TypeScript API

Plugins are distributed as WASM modules (for performance-critical tasks like custom DRC rules) or TypeScript bundles (for UI extensions). The plugin API exposes: layout database read/write, canvas drawing primitives, command registration, keybinding registration, panel creation, and simulation control. PDK definitions are themselves plugins that register layer maps, design rules, device models, and standard cell libraries. This means adding a new technology is a matter of installing a plugin, not modifying source code.

4.4 Layout Database: In-Memory EDA Kernel

The core data structure is a hierarchical cell database written in Rust, inspired by the OpenDB model used in OpenROAD. Each cell contains a flat list of geometric primitives (rectangles, polygons, paths, vias) organized by layer, plus a list of subcell references with transformations. The database

supports full undo/redo via a command-pattern journal, incremental DRC (only re-checking modified regions), and event-based notifications for the UI. All geometric operations (boolean, merge, size, clip) use the Clipper2 library via Rust FFI.

4.5 DRC Engine: Incremental Rule Checking

Unlike Microwind's monolithic DRC pass, OpenSilicon uses an incremental DRC engine. When the user modifies geometry, only the affected spatial region is re-checked. DRC rules are defined declaratively in a YAML-based rule deck that the PDK plugin provides. Rules support: minimum width, minimum spacing (both Manhattan and Euclidean), enclosure, extension, density, antenna, and custom programmable checks written in WASM. Violations are rendered as real-time overlays on the canvas with interactive markers that snap the user to the error location.

4.6 Simulation Integration

OpenSilicon includes a built-in parasitic extraction engine (RC extraction using the layout geometry and PDK interconnect models) and generates SPICE netlists. For simulation, rather than shipping a built-in simulator, it integrates with external engines via a plugin interface. The default plugin targets ngspice (open-source), but plugins can also target Xyce, Spectre, or cloud-based simulation services. Simulation results are displayed in an integrated waveform viewer panel with cursor tracking, measurements, and overlay comparison.

5. UI/UX Architecture

5.1 Workspace Layout

The workspace mirrors VSCode's proven layout: a left sidebar with file explorer, cell hierarchy browser, layer palette, and component library; a central editor area with tabbed layout canvases and split-view support; a right sidebar for properties, DRC violations, and measurement tools; and a bottom panel with terminal, simulation output, and waveform viewer. Every panel is draggable, resizable, and closable. The layout persists per-project.

5.2 Command Palette & Keybindings

Every action in the application (draw rectangle, run DRC, export GDS, toggle layer visibility, switch PDK) is registered as a named command. The command palette (Ctrl+Shift+P) provides fuzzy-search access to all commands. All keybindings are user-configurable via a JSON file, with presets for Microwind, Magic, KLayout, and Cadence Virtuoso for easy migration. Context-sensitive keymaps activate depending on whether the user is in the layout canvas, netlist editor, waveform viewer, or terminal.

5.3 Layer Palette & Visualization

The layer palette shows all technology layers with toggle visibility, toggle selectability, color/pattern picker, and opacity slider. Layers support fill patterns (solid, hatched, cross-hatched, stipple) that are rendered by the GPU shader. The palette also shows quick-access buttons for common layer combinations (e.g., 'Show only metal layers', 'Show diffusion + poly', 'Show all'). Layer colors are defined by the PDK plugin but user-overridable.

6. Development Roadmap

Phase 1: Core Foundation (Months 1–4)

Sprint	Deliverable	Owner	Status
1–2	Tauri app shell, React workspace with panel system, theming engine	Frontend	Planned
3–4	Rust layout database with cell hierarchy, undo/redo, GDS-II import/export	Backend	Planned
5–6	WebGPU canvas renderer: polygon drawing, layer coloring, zoom/pan, selection	Renderer	Planned
7–8	Basic editing tools: rectangle, polygon, path, via, move, copy, stretch, delete	Tools	Planned

Phase 2: Design Intelligence (Months 5–8)

Sprint	Deliverable	Owner	Status
9–10	Plugin system (WASM + TS API), PDK plugin format, SKY130 PDK plugin	Platform	Planned
11–12	Incremental DRC engine with YAML rule decks, real-time violation overlay	DRC	Planned
13–14	Parasitic extraction (RC), SPICE netlist generation, ngspice integration	Simulation	Planned
15–16	Command palette, configurable keybindings, keyboard-driven workflow	UX	Planned

Phase 3: Ecosystem & Collaboration (Months 9–12)

Sprint	Deliverable	Owner	Status
17–18	Git integration: diff view for layouts, branch/merge, commit history	VCS	Planned
19–20	Waveform viewer, simulation parameter sweeps, corner analysis	Simulation	Planned
21–22	Marketplace for plugins/PDKs, built-in package manager	Ecosystem	Planned
23–24	LVS (Layout vs Schematic), netlist comparison view, cross-probing	Verification	Planned

Phase 4: Production Readiness (Months 13–18)

Sprint	Deliverable	Owner	Status
25–28	Additional PDKs: GF180MCU, IHP SG13G2, ASAP7. Standard cell viewer/editor.	PDK Team	Planned
29–32	OpenROAD integration: place-and-route import, timing-annotated layout view	Integration	Planned
33–34	Collaborative editing (CRDT-based) for team design sessions	Collab	Planned

Sprint	Deliverable	Owner	Status
35–36	Tapeout CI/CD: automated DRC/LVS/antenna checks, GDS-II signoff pipeline	DevOps	Planned

7. Technology Stack

Layer	Technology	Rationale
App Framework	Tauri 2.x (Rust)	10x smaller than Electron, native performance, secure IPC
Frontend	React 19 + TypeScript	Component model, vast ecosystem, type safety
State Mgmt	Zustand + Immer	Simple, performant, supports undo/redo natively
Renderer	WebGPU (wgpu-rs)	GPU-accelerated 2D rendering, compute shaders for DRC
Layout DB	Custom Rust crate	Hierarchical cell DB with R-tree spatial index
Geometry	Clipper2 (Rust FFI)	Boolean ops, offset, minkowski for DRC checks
File Formats	gds-rs, oasis-rs, lef-def-rs	Native Rust parsers for GDS-II, OASIS, LEF/DEF
Plugin Runtime	Wasmtime + Deno	WASM for perf-critical plugins, Deno for TS/JS plugins
Simulation	ngspice (default), Xyce	Open-source SPICE engines via plugin interface
Testing	Vitest + Rust tests	Unit, integration, and visual regression testing
CI/CD	GitHub Actions	Cross-platform builds, auto-release, plugin marketplace CI

8. File Format Strategy

One of the biggest failures of Microwind is its proprietary .MSK binary format. OpenSilicon takes the opposite approach: the native project format is a directory containing human-readable files that work naturally with Git.

8.1 Project Structure

myproject.osproj/ project.json ← metadata, PDK reference, settings cells/ inverter.gds ← standard GDS-II per cell inverter.meta.json ← labels, pins, properties nand2.gds nand2.meta.json testbenches/ inv_tb.spice ← simulation netlists simulations/ inv_dc.raw ← simulation results rules/ drc_overrides.yaml ← project-specific DRC tweaks

Because each cell is a standard GDS-II file, users can open individual cells in KLayout, Magic, or any other GDS viewer. The metadata JSON files store only what GDS-II cannot: pin assignments, cell categories, and user notes. This directory-based format means Git can track changes at the cell level, diffs are meaningful, and merges work naturally.

9. Competitive Positioning

Feature	OpenSilicon	Microwind	KLayout	Magic	Virtuoso
GPU Rendering	✓	✗	Partial	✗	✓
Plugin System	✓	✗	✓ (Ruby)	✗	✓ (SKILL)
Cmd Palette	✓	✗	✗	✗	✗
Git Integration	✓	✗	✗	✗	✗
SPICE Sim	✓ (plugin)	Built-in	✗	✗	✓ (ADE)
Real PDKs	✓	✗	✓	✓	✓
Cross-Platform	✓	Win only	✓	Linux	Linux
Open Source	✓	✗	✓	✓	✗
Cost	Free	$200–$2000	Free	Free	$50k+/yr

10. Team Structure & Resource Requirements

10.1 Core Team (Phase 1–2)

Role	Count	Key Skills
Rust Systems Engineer	2	wgpu, spatial algorithms, EDA data structures, performance optimization
React/TS Frontend	2	Complex canvas UIs, Tauri IPC, state management, accessibility
EDA Domain Expert	1	DRC/LVS algorithms, SPICE extraction, PDK development, tapeout flow
UX/UI Designer	1	Developer tool UX, design systems, information density, dark mode
DevOps / Build	0.5	Cross-platform CI, Tauri packaging, auto-update system

10.2 Estimated Budget (18-month plan)

Category	Estimate (USD)
Engineering salaries (6.5 FTE × 18 months)	$1.5M – $2.2M
Cloud infrastructure (CI, testing, marketplace)	$30K – $50K
PDK licensing & shuttle runs for validation	$20K – $40K
Design, branding, documentation	$15K – $25K
Total	$1.6M – $2.3M

For an open-source approach with community contributors, the core team can be reduced to 3–4 people with a budget of $600K–$900K for the first year, focusing on Phases 1–2 only and relying on community contributions for PDK plugins and additional simulation backends.

11. Go-to-Market Strategy

11.1 Launch Strategy

- Alpha release (Month 6): Core editing + SKY130 PDK + basic DRC. Target 50 beta testers from university EDA labs.
- Beta release (Month 10): Full plugin system + simulation + Git integration. Open GitHub repository for community contributions.
- v1.0 release (Month 14): Production-ready with 3+ PDKs, marketplace, documentation, and tutorial series.
- University partnerships: Free academic licenses (it's open-source), teaching material packages, integration with popular VLSI courses.

11.2 Revenue Model (Optional)

- Core tool is free and open-source (Apache 2.0 or similar).
- Premium cloud features: hosted simulation farm, collaborative editing server, CI/CD tapeout pipeline — subscription-based.
- Enterprise support: Priority bug fixes, custom PDK development, on-premise deployment consulting.
- Marketplace revenue share: 15% cut on paid plugins/PDKs sold through the built-in marketplace.

12. Risks & Mitigations

Risk	Likelihood	Impact	Mitigation
WebGPU browser support	Medium	High	Fallback to WebGL2 renderer; Tauri uses native wgpu which doesn't depend on browser support
DRC accuracy vs commercial tools	High	High	Use Calibre/Pegasus as reference; automated regression suite comparing DRC results on test layouts
PDK availability/correctness	Medium	High	Partner with Google/Efabless for SKY130; use OpenPDK initiative; community-validated rulesets
Adoption inertia	High	Medium	VSCode familiarity reduces learning curve; Microwind keybinding preset; migration wizard for .MSK files
Performance at scale	Medium	Medium	Rust backend handles computation; R-tree spatial index; GPU instanced rendering; benchmark suite

Risk	Likelihood	Impact	Mitigation
Plugin security	Low	High	WASM sandboxing; permission model for file/network access; signed plugins for marketplace

13. Success Metrics

Metric	6-Month Target	18-Month Target
GitHub stars	1,000	10,000
Monthly active users	500	5,000
Community plugins	5	50+
Supported PDKs	1 (SKY130)	5+
University adoptions	3 courses	20+ courses
DRC rule coverage vs Calibre	80%	95%+
Canvas render performance	1M polygons @ 30fps	10M polygons @ 60fps

14. Immediate Next Steps

- Set up the monorepo: Rust workspace (core crate, renderer crate, DRC crate) + Tauri + React frontend. Toolchain: cargo + pnpm + turborepo.
- Build a proof-of-concept: Load a SKY130 GDS-II file, render it on a WebGPU canvas with layer colors, and implement zoom/pan. This demonstrates the core value proposition in 2–3 weeks.
- Design the plugin API contract: Define the TypeScript interfaces and WASM host functions before building the plugin runtime. Get community feedback on the API design.
- Write the PDK plugin spec: Document exactly what a PDK plugin must provide (layer map, DRC rules, device models, interconnect stack, standard cells) so the community can start contributing.
- Recruit 2–3 academic advisors: Professors who teach VLSI layout courses and can validate the feature set, provide feedback on educational workflows, and pilot the tool in their courses.

The goal is not to build another layout editor.

The goal is to build the platform that makes every future VLSI tool possible.